Université Savoie Mont Blanc,
UFR SceM
INFO505, L3 INFO

September 3, 2023
J.-O. Lachaud

Fiche 5 — Structures auto-référentes, pointeurs de fonctions, préprocesseur

Cette fiche explique comment représenter des structures de données évoluées en C, notamment les structures auto-référentes (listes, arbres, graphes), comment rendre le comportement des structures de données et des fonctions associées modifiable à l'exécution via les pointeurs de fonction. Quelques éléments d'utilisation du préprocesseur sont aussi donnés.

1 Structures auto-référentes

Les structures auto-référentes sont ubiquitaires en informatique. Ce sont des données typées en relation avec d'autres données du même type. L'exemple le plus général est la structure de graphe, et les listes ou les arbres sont d'autres cas particuliers. Ces structures apparaissent dans tous les domaines de l'informatique, des bases de données à l'informatique graphique en passant par les réseaux.

1.1 Déclaration des structures

En C, on ne peut utiliser dans la définition d'un type que des symboles définis auparavant. On utilise alors la syntaxe `struct S...*`, car une structure ne peut se contenir elle-même (elle serait de taille infinie). Une relation entre deux variables de ce type est donc naturellement matérialisé par un pointeur.

```c
// noeud d'une liste simplement chaînée
struct SFListNode {
  ... value;
  struct SFListNode* next;
};
typedef struct SFListNode FListNode;
// une liste est un pointeur vers son
// premier noeud (ou NULL si vide).
typedef FListNode* FList;
```

```c
// noeud d'un arbre binaire
struct SBinTreeNode {
  ... value;
  struct SBinTreeNode* left;
  struct SBinTreeNode* right;
};
typedef struct SBinTreeNode BinTreeNode;
// un arbre binaire est un pointeur vers
// sa racine (ou NULL si vide).
typedef BinTreeNode* BinTree;
```

Notez qu'en C, au contraire des langages fonctionnels, on distingue le type pour représenter chaque élément de la liste/arbre, du type qui désigne la liste/arbre tout entier.

On pourrait tout à fait faire des listes / arbres, sans allocation dynamique. Le problème est que le nombre de noeuds devrait être connu à l'avance : cela manque de souplesse. Je vous donne néanmoins un exemple de code ci-dessous:

```c
FListNode N1 = { 17, NULL };
FListNode N2 = { 12, &N1 };
FListNode N3 = { 13, &N2 };
FList L = &N3; // L = (13, 12, 17)
for ( FListNode* A = L; A != NULL;
      A = A->next )
  printf( " %d", A->value );
// Affiche : 13 12 17
```

```c
BinTreeNode N1 = { "7", NULL, NULL };
BinTreeNode N2 = { "3", NULL, NULL };
BinTreeNode N3 = { "4", NULL, NULL };
BinTreeNode N4 = { "*", &N2, &N3 };
BinTreeNode N5 = { "+", &N1, &N4 };
BinTree B = &N5;
// B = (+, 7, (*, 3, 4) ),
// i.e. 7 + 3 * 4
```

Question 1. Quel serait une structure pour un graphe qui relie des chaînes de caractères ? Un exemple est un ensemble de mots, liés s'ils apparaissent ensemble dans plus de 20% d'un corpus de textes.

Question 2. Même question mais on veut mettre une donnée (mettons un réel) sur chaque arête du graphe ? Un exemple est un réseau routier, avec les distances entre les noeuds/villes.

1.2 Fonctions de manipulation

On voit que la méthode précédente de liaison entre variables est trop rigide, et aussi sujette à erreur. Comme d'habitude en C, on se donne des fonctions de manipulation des structures de données. Ensuite, l'utilisateur les utilise et minimise donc le risque d'erreur. Comme on veut créer des listes arbitraires, nos fonctions vont utiliser l'allocation dynamique.

On prend l'exemple des listes simplement chaînées (*forward list*, ici la structure `FList` et ses noeuds `FListNode`).

```
/* FList.h */

// Type des valeurs stockées
typedef int Elem;

// Noeud d'une (forward) liste
struct SFListNode {
  Elem value;
  struct SFListNode* next;
};
typedef struct SFListNode FListNode;
// Une liste est un pointeur vers son
// premier noeud (ou NULL si vide).
typedef FListNode* FList;
// Un iterateur parcourt une liste
typedef FListNode* FIter;
```

```
// Initialisation, terminaison, vide ? ----
void  FList_init( FList* pL );
bool  FList_is_empty( FList* pL );
void  FList_finish( FList* pL );
// Parcours, accès aux élements ----------
FIter FList_begin( FList* pL );
FIter FList_end( FList* pL );
FIter FList_next( FList* pL, FIter it );
Elem  FList_value( FList* pL, FIter it );
void  FList_set_value( FList* pL, FIter it
    , Elem v )
// Insertion, suppression ----------------
FIter FList_insert_first( FList* pL, Elem
    v );
FIter FList_insert_after( FList* pL, FIter
    it, Elem v );
void  FList_erase_first( FList* pL );
void  FList_erase( FList* pL, FIter it );
```

Le code des fonctions est donné ci-dessous:

```
//-------------------------------------
// Initialisation, terminaison, vide ? ----
//-------------------------------------
// Initialise une variable liste *pL
void FList_init( FList* pL )
{ // NB: pL est un pointeur de pointeur !
  *pL = NULL; // la liste est mise à vide
}
// Retourne 'true' si et seulement si la
// liste est vide.
bool FList_is_empty( FList* pL )
{
  return *pL == NULL;
}
// Vide la liste *pL.
void FList_finish( FList* pL )
{ // NB: pL est un pointeur de pointeur !
  if ( FList_is_empty( pL ) ) return;
  FIter iter = FList_begin( pL );
  while ( FList_next( pL, iter )
          != FList_end( pL ) )
    FList_erase( pL, iter );
  FList_erase_first( pL );
}
```

```
//-------------------------------------
// Parcours, accès aux élements ----------
//-------------------------------------

// Retourne un iterator sur le premier
// élément
FIter FList_begin( FList* pL )
{ return *pL; }
// Retourne un iterator sur l'élément
// suivant à it dans la liste *pL
FIter FList_next( FList* pL, FIter it )
{ return it->next; }
// Retourne la valeur de l'élément pointé
// par l'iterateur it
Elem FList_value( FList* pL, FIter it )
{ return it->value; }
// Change la valeur de l'élément pointé
    par
// l'iterateur it
void FList_set_value( FList* pL, FIter it,
                      Elem v )
{ it->value = v; }
// Retourne un iterator (invalide)
    pointant
// après le dernier élément
FIter FList_end( FList* pL )
{ return NULL; }
```

```
//-------------------------------------------
// Insertion, suppression
//-------------------------------------------

// Insère la valeur v au début de *pL
FIter FList_insert_first
  ( FList* pL, Elem v )
{
  FListNode* elem = (FListNode*)
    malloc( sizeof( FListNode ) );
  elem->value      = v;
  elem->next       = *pL;
  *pL              = elem;
  return elem;
}
// Insère la valeur v après la valeur
// pointée par it
FIter FList_insert_after
  ( FList* pL, FIter it, Elem v )
{
  FListNode* elem = (FListNode*)
    malloc( sizeof( FListNode ) );
  elem->value      = v;
  elem->next       = it->next;
  it->next         = elem;
  return elem;
}
```

```
// Supprime le premier element de *pL
void FList_erase_first( FList* pL )
{
  assert( ! FList_is_empty( pL ) );
  FIter next = (*pL)->next;
  free( *pL );
  *pL = next;
}
// Supprime l'élément de *pL après it
void FList_erase( FList* pL, FIter it )
{
  FIter next = it->next;
  assert( next != NULL );
  it->next = next->next;
  free( next );
}
```

On voit que l'allocation ou la désallocation de noeuds se fait naturellement dans les fonctions d'insertion et de suppression de valeurs. Parcourir une liste, accéder aux éléments, ou même changer une valeur dans la liste n'induit pas d'allocation dynamique.

Question 3. La différenciation entre insérer/supprimer en tête ou après un élément est obligatoire dans ce choix d'implémentation. Une autre solution aurait été de créer un premier élément *fictif*. Une liste n'est alors plus un pointeur sur un premier noeud mais *est* un noeud (dont la valeur est ignorée). Proposez une implémentation de ces listes, en les faisant de plus doublement chaînées (ou bidirectionnelles).

2 Pointeurs de fonction

Le nom d'une fonction est assimilable à l'adresse de son code source. On peut donc considérer un nom de fonction comme un pointeur, dont le type est ainsi formé:

```
TypeRetour NomFonction( Type1 param1, Type2 param2 );
typedef TypeRetour (*TypeFonction)( Type1 param1, Type2 param2 );
```

`TypeFonction` désigne alors un pointeur vers une fonction qui a deux paramètres en entrée (de types `Param1` et `Param2`), et qui retourne un `TypeRetour`. On peut par exemple faire ceci:

```
double addition( double x, double y )
{
  return x+y;
}
double multiplication( double x, double y )
{
  return x*y;
}
typedef double (*OperationBinaire)( double x, double y ); /* x,y facultatifs */

int main( void )
{
  double a = 10.0;
  double b = 0.5;
  OperationBinaire op = addition; /* &addition idem */
  printf( "%g\n", op(a,b) ); /* affiche 10.5 */
  op = multiplication; /* &multiplication idem */
  printf( "%g\n", op(a,b) ); /* affiche 5.0 */
  return 0;
}
```

Les pointeurs de fonction forment en C la brique de base pour modifier le comportement des structures de données, pour avoir des réactions à des événements/signaux personnalisés, plus généralement pour implémenter le polymorphisme.

Quelques exemples d'utilisation

- Définir une fonction *callback* pour une structure. C'est très utilisé en IHM, où on associe aux élément graphiques (fenêtre, zone de dessin, etc) des fonctions qui sont appelées pour un événement précis (clic souris, déplacement de la souris, ordre de réaffichage). On verra leur utilisation dans GTK dans le TP GTK Tetris

- Appliquer une fonction à une collection d'éléments. Nous donnons ci-dessous quelques exemples:

```
#include <stdlib.h>
#include <stdio.h>

typedef void (* FctOnDouble)( double* );
void reset( double* x )
{ *x = 0.0; }
void uniform( double* x )
{ *x = (double) rand() / (double) RAND_MAX; }
void add1( double* x )
{ *x += 1; }
void square( double* x )
{ *x *= *x; }
void affiche( double*x )
{ printf(" %f", *x ); }
void apply( double* begin, double* end, FctOnDouble f )
{
  for ( ; begin != end; ++begin )
    f( begin );
}
int main()
{
  double t[ 5 ];
  apply( t, t+5, reset );
  apply( t, t+3, uniform );
  apply( t+2, t+5, add1 );
  apply( t+4, t+5, add1 );
```

```
    apply( t, t+5, square );
    apply( t, t+5, affiche );
    printf("\n");
    return 0;
}
```

Question 4. Comment faire un accumulateur, un moyenneur ? Que faut-il rajouter en plus ?

Question 5. On veut faire une collection d'animaux, avec chacun un cri différent: le lion fait "Groaar !@!", la souris "squeak squeak", le chameau "blah blah", le chat "miaou ?", le chien "whaff !". Proposez une solution basée pointeur de fonction, telle que:

```
...
Animal t[ 5 ];
...
for ( int i = 0; i < 5; i++ )
  t[ i ].cri();
```

affiche les cris des différents animaux.

3 Le préprocesseur

Avant d'être compilé, le fichier source est transformé en un autre fichier texte (plus long en général) par ce qu'on appelle le préprocesseur. Celui-ci va détecter et interpréter les commandes commençant par '#'. Ensuite, ces commandes/définitions influent sur la sortie du préprocesseur. Les commandes les plus importantes sont :

`#include` ***file-name*** Dis au préprocesseur de copier/coller le fichier *file-name* dans ce source à cet endroit-là.

`#define` ***name value*** Dis au préprocesseur de définir une variable de substitution. Les variables de substitution du préprocesseur fournissent un moyen simple de nommer des constantes. En effet :

```
#define CONSTANTE valeur
```

permet de substituer presque partout dans le code source qui suit cette ligne la suite de caractères "CONSTANTE" par la valeur. Plus précisément, la substitution se fait partout, à l'exception des caractères et des chaines de caractères. Par exemple, dans le code suivant :

```
#define TAILLE 100
printf("La constante TAILLE vaut %d\n", TAILLE);
```

La substitution se fera sur la deuxième occurrence de TAILLE, mais pas la première. Le préprocesseur transformera ainsi l'appel à printf :

```
printf("La constante TAILLE vaut %d\n", 100);
```

Le préprocesseur procède à des traitements sur le code source, sans avoir de connaissance sur la structure de votre programme. Dans le cas des variables de substitution, il ne sait faire qu'un remplacement de texte, comme le ferait un traitement de texte. On peut ainsi les utiliser pour n'importe quoi, des constantes, des expressions, voire du code plus complexe.

`#define` ***macro-name(...) expr*** Une macro est en fait une constante qui peut prendre un certain nombre d'arguments. Les arguments sont placés entre parenthèses après le nom de la macro sans espaces, par exemple :

```
#define MAX(x,y) x > y ? x : y
#define SWAP(x,y) x ^= y, y ^= x, x ^= y
```

La première macro prend deux arguments et "retourne" le maximum entre les deux. La deuxième est plus subtile, elle échange la valeur des deux arguments (qui doivent être des variables entières), sans passer par une variable temporaire, et ce avec le même nombre d'opérations.

`#if ... #else ... #endif` Permettent la compilation conditionnelle. Ainsi

```
#if defined( DEBUG )
  assert( x > 0 );
#endif
```

Permet de ne vérifier la valeur de x que dans le mode `DEBUG`. On peut faire plus concis et plus pratique sous la forme suivante :

```
#if defined( DEBUG )
#define ASSERT( e ) assert( e )
#else
#define ASSERT( e ) /* vide ! */
#endif
...
  ASSERT( x > 0 ); /* ASSERT sera substitué automatiquement par
                      une instruction vide ou la fonction assert
                      selon que DEBUG est défini ou non. */
```

La commande `#ifdef` *name* est équivalente à `#if defined(` *name* `)`

On se sert beaucoup de ces commandes pour éviter qu'un fichier en-tête ne soit inclus plusieurs fois lors de la compilation d'un fichier source.

`#pragma` Appelle des commandes spécifiques, pas forcément partagées par tous les compilateurs. Cela permet parfois de supprimer des "warning"s de compilation que vous savez sans conséquences. On utilise beaucoup le pragma suivant en début de fichier entête pour éviter de l'inclure plusieurs fois.

```
#pragma once
```

Question 6. On reprend l'exemple de la macro `MAX`. Est-ce que vous obtiendrez le résultat attendu avec les lignes suivantes ?

```
#include <stdio.h>
#define MAX(x,y) x > y ? x : y
int main()
{
   printf( "MAX(3,5)=%d\n", MAX(3,5) );
   printf( "MAX(MAX(3,8),5)=%d\n", MAX(MAX(3,8),5) );
   printf( "MAX(5,MAX(3,8))=%d\n", MAX(5,MAX(3,8)) );
   printf( "4+MAX(3,5)=%d\n", 4+MAX(3,5) );
   return 0;
}
```

Voyez-vous un moyen de corriger ce problème ?